Levenshtein Edit Distance

The Levenshtein Edit Distance, also known as the minimum edit distance, is a metric used to measure the similarity between two strings. It calculates the minimum number of operations required to transform one string into another, where each operation can be an insertion, deletion, or substitution of a single character.

Given two strings, let's call them `source` and `target`, the Levenshtein Edit Distance can be calculated using a dynamic programming approach. We can build a matrix of size(m+1) x(n+1), where m and n are the lengths of the source and target strings, respectively. The element at position(i, j) in the matrix represents the minimum number of operations required to transform the first i characters of the source string into the first j characters of the target string.

The following recursive formula can be used to fill in the matrix:

```
D[i][j] =
    | 0 if i = 0 and j = 0
    | i if j = 0 (deleting i characters)
    | j if i = 0 (inserting j characters)
    | min(D[i-1][j] + 1,) if i > 0 and j > 0 (deletion)
    | min(D[i][j-1] + 1,)
    | min(D[i-1][j-1] + substitution_cost)
```

In the above formula, `substitution_cost` is 0 if the i-th character of the source string is equal to the j-th character of the target string, and 1 otherwise.

The final value in the bottom right corner of the matrix ($D[m][n]$) represents the Levenshtein Edit Distance between the source and target strings.

Now, let's move on to the pseudocode implementation.

Pseudocode

```
function levenshteinDistance(source, target):
    m = length(source)
    n = length(target)
    D = array of size (m+1) x (n+1)

    for i from 0 to m:
        D[i][0] = i

    for j from 0 to n:
        D[0][j] = j

    for i from 1 to m:
        for j from 1 to n:
            substitution_cost = 1
            if source[i-1] = target[j-1]:
                substitution_cost = 0
            D[i][j] = min(D[i-1][j] + 1, D[i][j-1] + 1,
                          D[i-1][j-1] + substitution_cost)

    return D[m][n]
```

The above pseudocode initializes the matrix `D` with appropriate values and then fills in the matrix using the dynamic programming approach. Finally, it returns the Levenshtein Edit Distance between the source and target strings.

Step-by-Step Explanation of Pseudocode

1. Initialize variables:
    - Set `m` as the length of the `source` string.
    - Set `n` as the length of the `target` string.
    - Create a matrix `D` of size `(m+1) x (n+1)` to store the minimum edit distances.
2. Fill in the base cases:
    - Set `D[i][0] = i` for each `i` from 0 to `m`, representing the cost of deleting `i` characters from the `source` string.
    - Set `D[0][j] = j` for each `j` from 0 to `n`, representing the cost of inserting `j` characters into the `source` string.
3. Compute the edit distances:
    - Iterate through `i` from 1 to `m` and `j` from 1 to `n`.
    - Calculate the `substitution_cost` as 1 if the `i`-th character of `source` is not equal to the `j`-th character of `target`, or 0 otherwise.
    - Update `D[i][j]` by taking the minimum value among the following:
        - `D[i-1][j] + 1`: Cost of deleting the `i`-th character from `source`.
        - `D[i][j-1] + 1`: Cost of inserting the `j`-th character into `source`.
        - `D[i-1][j-1] + substitution_cost`: Cost of substituting the `i`-th character of `source` with the `j`-th character of `target`.
4. Return the final edit distance:
    - The value at `D[m][n]` represents the minimum edit distance between the `source` and `target` strings.

Time Complexity Analysis

The time complexity of the Levenshtein Edit Distance algorithm can be analyzed as follows:

- Initializing the matrix `D` takes constant time, O(1).
- Filling in the base cases requires iterating through `m` and `n`, resulting in O(m + n) time complexity.
- Computing the edit distances involves nested iterations over `m` and `n`, resulting in a time complexity of O(m * n).
- Overall, the time complexity of the algorithm is O(m * n).

Python Code

```python
def levenshtein_distance(source, target):
    m = len(source)
    n = len(target)
    D = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        D[i][0] = i

    for j in range(n + 1):
        D[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            substitution_cost = 1
            if source[i - 1] == target[j - 1]:
                substitution_cost = 0
            D[i][j] = min(D[i - 1][j] + 1, D[i][j - 1] + 1, D[i - 1][j - 1] + substitution_c

    return D[m][n]
```

The `levenshtein_distance` function above takes two strings, `source` and `target`, as input and returns the Levenshtein Edit Distance between them.

You can call the function like this:

```python
source = "kitten"
target = "sitting"
distance = levenshtein_distance(source, target)
print(distance)  # Output: 3
```